Transition: DS+ Algorithms

Backtracking (pt 2)

Worksheet:

code:

```
⟨⟨Does any subset of X[1..i] sum to T?⟩⟩
SUBSETSUM(X, i, T):
  if T = 0
      return TRUE
  else if T < 0 or i = 0
      return FALSE
  else
      with ← SUBSETSUM(X, i − 1, T − X[i])    ⟨⟨Recurse!⟩⟩
      wout ← SUBSETSUM(X, i − 1, T)           ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

Text Segmentation

Fix a "language", so can recognize "words".

Ex: - English text : return true if word in dict } O(1)

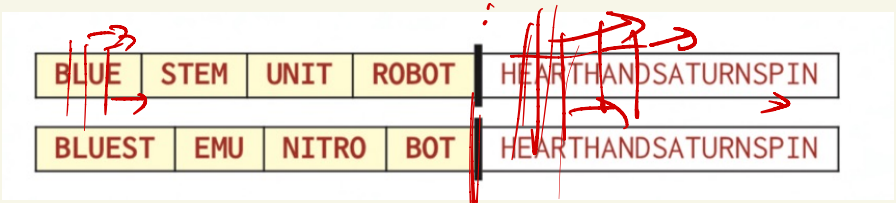
- palindromes :

- genetic data : CGGTA---
  ↳ find break down of : O(1)
  relevant codes

⇒ Subroutine IsWord(s) will be given
↳ O(1)

Q: What happens to a smaller word that overlaps or is later?

Ex:



ignore
trust
A[1..i-1] is
already
parsed

i
HE
HEAR
HEART
HEARTH

Choose

try all of them

Code:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

O(1)

try all next words

recurse on rest (if it's a word)

IsWord is called O(n) times

Runtime: $S(n)$ = time this takes on string of length n

$\Rightarrow \leq O(n) + S(n-1) + S(n-2) + S(n-3) \ldots S(1) = \sum_{i=1}^{n} S(i) + O(n)$

Issue w/ passing arrays: space (plus annoying!)

isWord : $O(1)$

↳ c for some constant c

for $i \leftarrow 1$ to n :

    isWord

$$\sum_{i=1}^{n} c = c \cdot n = \Theta(n)$$

Passing by index / ptr / global / etc

Given an index i, find a segmentation of the suffix $A[i..n]$.

Formalize an (ugly?) recursion:

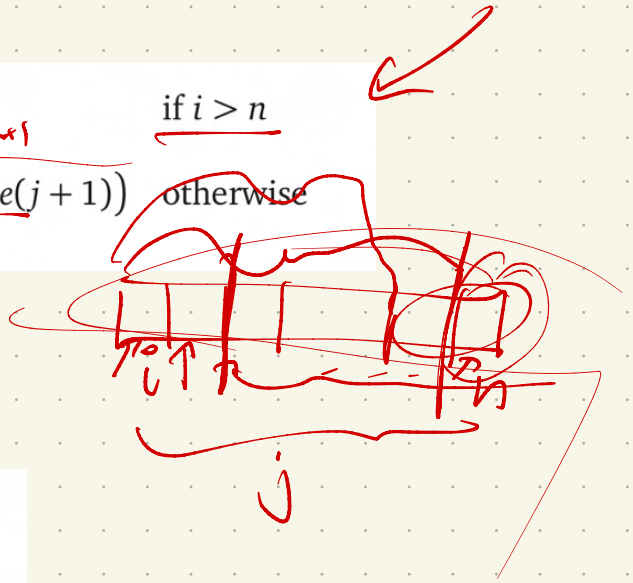
$$Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(\text{IsWord}(i,j) \wedge Splittable(j+1)\big) & \text{otherwise} \end{cases}$$

1..n

i+1

or

and



& then code it:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

index

Longest Increasing Subsequence

Ex: 8, 3, 5, 1, 6, 2, 7, 4, 9, 0, 10, 6

8 < 9 < 10 → increasing

3

Find LIS:

size 6

Ideas:

- take smallest

Pick → fails

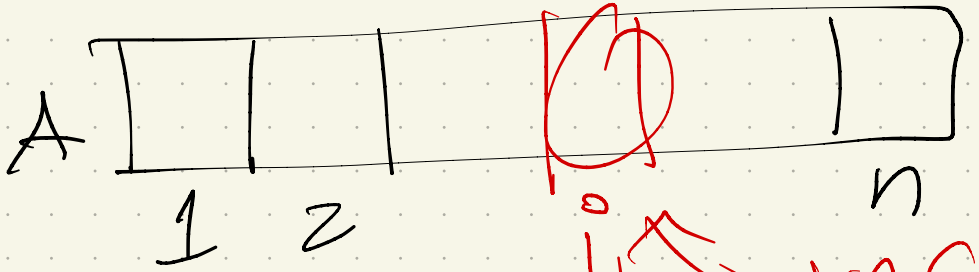
greed

- take first

→ fails

Why "jump to the middle"?
I Need a recursion!

First: how many subsequences?



i ← include i, or don't

↳ 2^n subsequences

Backtracking approach:

At index i: need last element included.

include i only if A[i] > last elt

Result:

Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

Example:

8	3	5	1	6	2	7	4	9	0	10	6
1	2	3	4	5	6	7	8	9	10	11	12

i j

Recursion:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

$j = 5$

include 6: $A[5] = 6$

$LISbigger(5, 6) = LISBigger(5, 7)$ (skip 6)

$LISbigger(5, 7) =$ try with $A[7] = 7$: $1 + LIS(7, 8)$

try without : $LIS(5, 8)$

Code version:

helper recursion

```
LISBIGGER(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISBIGGER(i, j + 1)
  else
      skip ← LISBIGGER(i, j + 1)
      take ← LISBIGGER(j, j + 1) + 1
      return max{skip, take}
```

if A[j] is too small, move on

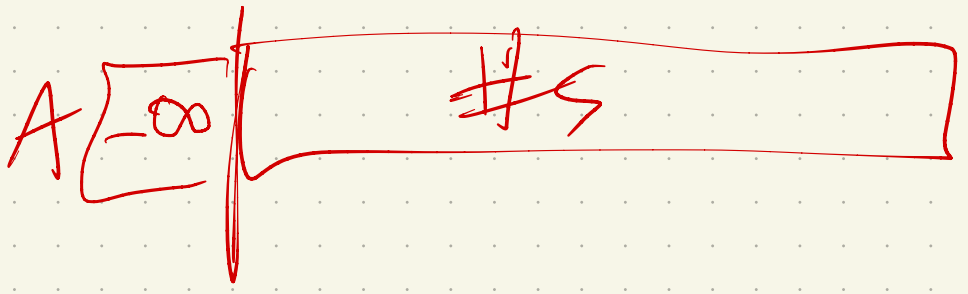
try both with A[j] & without

keep best

Problem – what did we want??

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```

return length of Longest Inc Sub in A

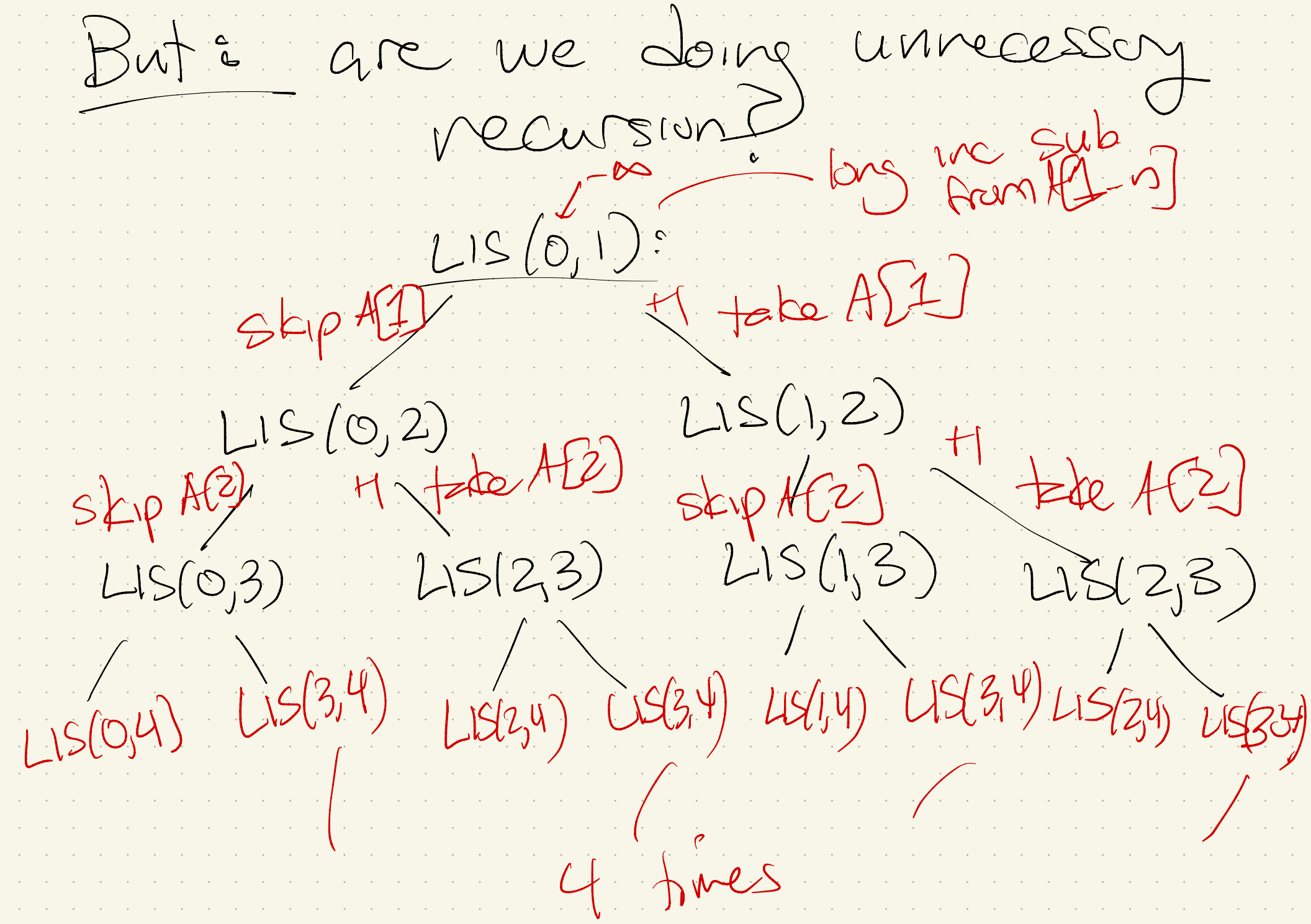


Runtime:

$$T(n) \leq 2T(n-1) + O(1)$$

$$\Rightarrow O(2^n)$$

BAD


But : are we doing unnecessary recursion?
-∞
long inc sub from A[1-n]
LIS(0,1):
skip A[1]
+1 take A[1]
LIS(0,2)
LIS(1,2)
skip A[2]
+1 take A[2]
skip A[2]
+1 take A[2]
LIS(0,3)
LIS(2,3)
LIS(1,3)
LIS(2,3)
LIS(0,4)
LIS(3,4)
LIS(2,4)
LIS(3,4)
LIS(1,4)
LIS(3,4)
LIS(2,4)
LIS(3,4)
4 times